

Serial No.: 09/436,618
Conf. No.: 6893

Art Unit: 2127

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant: PARKES et al.
Serial No: 09/436,618
Confirmation No: 6893
Filed: November 9, 1999
For: A METHOD OF PIPELINED PROCESSING OF PROGRAM DATA

Examiner: Ali, Syed J.
Art Unit: 2127

CERTIFICATE OF MAILING UNDER 37 C.F.R. § 1.8(A)

The undersigned hereby certifies that this document is being placed in the United States mail with first-class postage attached, addressed to Mail Stop AMENDMENT Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450, on the 29th day of Oct, 2004.


Signed:

Commissioner for Patents – MAIL STOP AMENDMENT
P.O. Box 1450
Alexandria VA 22313-1450

INVENTOR DECLARATION OF
MICHAEL A. B. PARKES and FREDERIC O. VICIK
UNDER 37 C.F.R. § 1.131

We, Michael A. B. Parkes and Frederic O. Vicik, declare that:

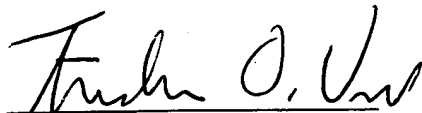
1. We are the inventors of the invention claimed in claims 1-38 recited in U.S. Patent Application Serial No. 09/436,618, filed November 9, 1999, and titled A METHOD OF PIPELINED PROCESSING OF PROGRAMMED DATA [hereinafter Present Application].
2. With reference to the invention claimed in the Present Application, we reduced the invention to practice at least as early as September 9, 1997. Specifically, we wrote a paper enabling the claimed invention and submitted that paper for publication at an Association for Computing Machinery SIGMETRICS conference, to be held June, 1998. A copy of this paper is provided as Exhibit A. However, after a confidential review, this paper was rejected for publication on February 3, 1998.

3. The paper submission process for the SIGMETRICS conference is confidential.
4. All work leading to the Present Application took place in Redmond, Washington of the United States, and occurred at least as early as September 9, 1998.
5. At the time of the development and reduction to practice of the invention, we were employees of Microsoft Corporation.

All of the above statements are made of my own knowledge are true, and all statements made on information and belief are believed to be true. We are aware that willful false statements and the like are punishable by fine or imprisonment or both (18 U.S.C. § 1001), and may jeopardize the validity of the above-reference application or any patent issuing therefrom.

Michael A.B. Parkes

Date



Frederic O. Vicik

10/27/04

Date

Serial No.: 09/436,618
Conf. No.: 6893

Art Unit: 2127

3. The paper submission process for the SIGMETRICS conference is confidential.
4. All work leading to the Present Application took place in Redmond, Washington of the United States, and occurred at least as early as September 9, 1998.
5. At the time of the development and reduction to practice of the invention, we were employees of Microsoft Corporation.

All of the above statements are made of my own knowledge are true, and all statements made on information and belief are believed to be true. We are aware that willful false statements and the like are punishable by fine or imprisonment or both (18 U.S.C. § 1001), and may jeopardize the validity of the above-reference application or any patent issuing there from.

Michael A.B. Parkes
Michael A.B. Parkes

10/27/98
Date

Frederic O. Vicik

Date

An New Architecture for High Performance Servers

Michael Parkes and Rick Vicik
Microsoft Visual C++ and SQL Server
1 Microsoft Way
Redmond, WA, USA.
September 9th, 1997

Abstract

The past decade has seen significant changes in computer hardware architectures coupled with massive improvements in aggregate system capacity and throughput. In this paper we would like to discuss some of these changes and make the case for a radically different software architecture aimed at making more efficient use of modern hardware. Specifically, we would like to focus our attention on improving application performance by enhancing spatial locality and simplifying the management of parallelism in server oriented applications.

1. The Gathering Storm

The merchants of doom have been howling for some time that the increases in hardware performance can not be sustained in the long term. Their warnings have merit as the laws of physics clearly indicate we are approaching physical boundaries that will be very hard to cross. Moreover, there are already significant performance issues on the horizon that will have to be overcome in the next few years. Let's us take a closer look at two of these issues namely: spatial locality and parallelism.

The software community has quietly ridden the dramatic improvements in hardware performance over the last decade and, to some extent, has come to expect more of the same. These performance increases have been largely painless and have forced few (if any) significant changes in the way software is typically developed. Certainly, if we consider that less than a decade ago an Intel 386 running at 16Mhz was considered an acceptable processor, we can see how large the improvement has been over this relatively short period of time. Amusingly enough, the brave hearted can still actually load

and execute some of the more recent software titles onto such a system and experience what life would have been like without the 486, Pentium and Pentium Pro. However, there is evidence to suggest that the free ride may well be over and that further performance improvements will soon require software modifications. Worst than this, these changes will probably be architectural in nature.

Let's consider one of Intel's recent offerings, namely: the 200Mhz Pentium Pro, and examine some of its basic metrics. The configuration we shall examine is a 4-way 200Mhz Pentium Pro with a 512k L2 cache and 50ns DRAM. According to information supplied by Intel, this processor can execute 3 instructions every clock. Practically, the sustainable rate is somewhat lower. Our personal measurements indicate that around 1 Clock Per Instruction (CPI) is a more realistic maximum for non-trivial code. We propose to use 1 CPI as our approximate baseline figure. Let us additionally assume that we generate one code reference per instruction executed plus an additional data reference every four instructions executed. Thus, we can calculate that we would expect to see about 200M code and 50M data memory references per second (i.e. $200M \times 2 \text{ bytes} + 50M \times 4 \text{ bytes} = 600MB \text{ per second}$). Now let us consider how this memory reference stream will be satisfied. The first source of supply is the L1 cache. The L1 cache in the Pentium Pro runs at full speed and can easily keep up with the 250M memory references (i.e. 200M code and 50M data) made per second. However, as its capacity is only 8k+8k (256 cache lines \times 2) a CPU can easily consume its contents in microseconds. The L2 cache is the next source of supply and is either 256k, 512k or 1M (i.e. 8k to 32k cache lines) with a 4-1-1-1 latency. The 4-1-1-1 latency means that it takes the L2 cache 4 clocks to produce the first 64 bits of the cache line being requested and 1 clock to produce each of the

following 64 bits (giving a total of 32 bytes which is the cache line size). This 4-1-1-1 latency means that the L2 can only cope with a miss every 7 clocks (i.e. $4+1+1+1 = 7$) or a 14% miss rate (i.e. 36M cache fills per second). This is quite acceptable given that each Pentium Pro CPU has its own private L2 cache and uses speculative execution to hide the L2 latency as much as possible. The final source of supply is main memory but here the metrics are significantly higher. Although each Pentium Pro runs at 200Mhz internally it only runs at 66Mhz externally. Hence, every external clock cycle takes 3 internal clock cycles. So if the approximate latency of accessing a DRAM is 5-1-1-1 external clocks this means 24 internal clocks (i.e. $(5+1+1+1) * 3 = 24$). At this speed we can only afford to miss the L1 and L2 caches once every 24 clocks or a miserly 4% miss rate (i.e. 8M cache fills per second). As if this were not concerning enough, this bandwidth has to be shared between all the available CPUs. Fortunately, the Pentium Pro supports a split transaction bus so in the presence of multiple requests it can supply a maximum aggregate throughput of 14M cache fills per second. Clearly, this can easily be a severe bottleneck and can cause CPUs to stall repeatedly and for significant periods of time. Moreover, measurements taken with Microsoft SQL Server indicated average latencies can rise to around 150 clocks per cache fill under a heavy load (such as a well known order entry benchmark). It is very hard to hide these kinds of latencies and consequentially most systems perform and scale badly under these types of conditions.

Historically, the performance increases in external bus and DRAM sub-systems have not kept pace with the improvements in CPU technology. If this trend continues some way will need to be found to bridge this increasing disparity in performance. To date, caches have served us very well but as can be seen from the previous narrative, they are fast approaching breaking point. Certainly, there are still some choices available at this time (i.e. bigger caches, faster buses) but whether they will be enough to support the next 2 or 3 generations of CPUs is a difficult question to answer. Certainly, preliminary information indicates that we can expect a typical latency of 100 clocks to main memory within 2 years.

Finally, before moving on, we would like to demonstrate how serious this type of problem

could be, even for a very simple program, with the aid of a trivial example. Consider a how a 200Mhz Pentium Pro might execute a portion of straight-line code consisting of 1 Million iterations of the operations: Load integer, Jump if integer is zero. Assume the value is present in the L1 cache and the jump is never taken. In this case we should be able to execute this sequence at full speed. If the value has to be fetched from the L2 cache then this will not hurt performance as long as the miss rate is less than 14%. If the value needs to be fetch from RAM then the miss rate must be less than 4%. If either of these limits is exceed performance starts to fall off rapidly as shown in the graph. A miss rate of just 20% on the L1 cache reduces performance by almost 30% and the same miss rate on the L2 cache reduces performance by almost 80%. Moreover, this data is a little conservative as it ignores real world issues such as an uneven miss rates (i.e. bursts of misses) and mis-predicted jumps. These issues only tend to make matters worse (much worse). In summary, this begs a satirical question about what one might call a 200Mhz Pentium Pro running at an effective rate of 40Mhz. May we suggest "a decent 486".

2. Building a Better Mouse Trap

We have seen that the ineffective use of caches on common commercial microprocessors (i.e., the Intel Pentium Pro) can seriously degrade performance. Nonetheless, this is a very difficult problem to overcome. Let us consider common software engineering practice and consider what (if anything) is wrong with it.

The traditional model of Symmetric Multi-Processor (SMP) programming uses a single thread of execution to perform an entire task (or significant sub-task) from start to finish. As a practical example, the execution of a transaction in Microsoft SQL Server 6.5 is currently performed from start to finish by a single worker thread. When multiple clients execute concurrent transactions a single thread for each client runs within the database engine to complete its assigned transaction. We believe most engineers are familiar with this model and that (with some minor variations) we could call it the industry standard model.

2.1. An Analogy

Let us digress and consider the methods used in decades past to manufacture cars. During this period often a single person (or a small team) manufactured a car by collecting the parts as needed and slowly assembling the pieces. A similar method to this is still used by Rolls Royce and yields a product second to none in terms of quality. However, reflecting on our previous discussion we can see that the spatial locality properties of this method of construction are very poor. It is simply not possible to have all the parts that are needed nearby (i.e. within arms length) and thus productivity (i.e. performance) is negatively affected. This is a quaint and a pleasant way to assemble products but it is certainly not efficient. In the car industry what followed was a complete redesign of manufacturing methodologies and today almost all cars are assembled on a production line using a totally different technique.

A modern assembly plant goes out of its way to optimize operations and enhance spatial locality. Its goal is to maximize throughput (i.e. performance) by all reasonable means. The basic idea is this: have a large number of simple steps and optimize each step as much as possible. Make sure that only the minimal amount of effort is expended to do each operation. Ensure efficiency and good spatial locality by surrounding each operator with all the parts they need so each iteration of the operation is fluid and quickly completed. Many consider this model the paragon of efficiency for mass production. We intend to borrow it to form the basis of the software architecture proposed in this paper.

At this point we hope that we have at least captured the reader's interest but expect that there are a number of very serious questions floating around relating to how such a model could be made to work for software development. Clearly, from what has gone before, it can be seen that the proposal is likely to be radical and unfamiliar to most practitioners. Hence, in the next section, we will explain our suggestion starting from familiar ground. However, as there is significant ground to cover, we request that the reader hold objections until sufficient detail has been outlined.

2.2. Nothing New under the Sun

We begin our description by considering the structure of a production server-oriented application. For convenience and consistency we will present Microsoft SQL Server 6.5 as our example. We will first examine how it executes a transaction at an abstract level and what types of data it uses. We note that, at a very coarse grain level, we can partition the data referenced into two main groups: current client state and global database state. The current client state contains information about the client who initiated a transaction, such as: user-id, network buffers, execution plan, open-table info (current position, search arguments, qualifying RIDs, pinned buffers), open transactions, heads of lock-ownership chains, program call-stack and the like. The global database state on the other hand contains information about the overall state of the database, that is: the heads of various non-private lists (page free list, lock free list, other free lists, page hash array, lock hash arrays, table hash array), the page cache (LRU list, dirty list, hash-synonym chain, checkpoint and lazy write functions), first/last/index-root page for each table, row count, current log buffer, unflushed log buffers (to guarantee Write-Ahead-Logging) and the like. Typically, the client state is significantly smaller than the global database state.

Now let us consider how this collection of data is used when we execute a transaction. In Microsoft SQL Server 6.5 a single thread collects a waiting transaction and step by step executes each stage of the transaction. This work requires the thread to obtain any necessary global database resources (and the related global data) and release them when it has finished. Generally, a thread runs on a particular CPU. Thus, on a Symmetric Multi-Processor (SMP) system the global database resources (and related global data) tend to move between CPUs. The client state is related to the thread and therefore tends to stay on a same CPU. Thus, a worker thread tends to optimize the caching of the current client state but not the global database state. This would seem reasonable as clearly either the client state or the global database state must move between CPUs. The diagram on this page outlines the point.

Unfortunately, a number of experiments we have conducted indicate that reality is somewhat more complex than the simple diagram above would

portray. Hard data (from internal tools) indicates that worker threads typically spend 95% of their time waiting for resources and only 5% of their time executing on a CPU. These resource latencies vary but are typically dominated by physical disk reads and log writes (i.e., around 75% of the overall elapse time spent executing the transaction). Additional data indicates that typically 20 other worker threads execute on a CPU during the time it takes to complete a physical I/O. Thus, as would be expected, these other threads tend to stomp all over the L1 and L2 caches destroying all traces of a waiting threads state. Even the top of a waiting thread's stack can be lost. Consequentially, it is not surprising that when a sleeping thread awakes it typically takes repeated back to back cache misses trying to reload its state into the caches. To compound this problem, applications like databases commonly contain a toxic mix of random I/O followed by a small amount of computation and then more random I/O. The effects of this can be crippling.

There is sometimes a tendency to blame synchronous operations like locks and I/O and imply that they are bad in some way. This is not the case, it is not these operations themselves that are the problem, but rather, the effects of the context switching they provoke. Whenever a CPU context switches, the current program's spatial locality is totally destroyed and the processor suddenly finds that its L1 cache is largely useless, the L2 cache is mostly useless and it has been asked to jump to some new portion of code totally unrelated to the previous one. This is pure poison to caches and it can easily take around 6,000 clocks for them to be reloaded.

This issue is at the heart of why Microsoft SQL Server 6.5 currently executes at around 4 CPI on a 200Mhz Pentium Pro quad processor machine when the core of each CPU is capable of sustaining at least 1 CPI. This is a four-fold loss in performance.

Let us now correct our previous diagram by showing that threads tend to execute for small amounts of time followed by long periods of inactivity.

Figure 3

Examining this diagram allows us to pose a question. What would happen if we changed

things around and instead we tried to optimize the global database state at the expense of the client state. This could be achieved by queuing requests on specific database resources and later executing all these requests in a succession. We can envisage this as a simple event loop (i.e., much like GUI programming) where the traditional events become pending requests to carry out specific database operations. We would clearly need multiple event loops, each one being optimized for some specific database operation, such as: buffer management, scanning index pages, scanning data pages, logging, I/O, sorting, updates and locks. Let us call one of these event loops a server and examine how it might operate in further detail.

At its most simplistic level, a server would consist of a queue of pending work, an event loop to process the work, and some portion of the global database state. When a server was called it would take each request and perform the required operations, returning the results to the caller. If a server was unable to complete an operation it would proceed as far as it could and then queue a request on some other server (as needed). Once this request had been queued it would be safe to simply move on to the next pending work item. The return mechanism requires that when a target server completes a request that it queues the original request packet back onto the calling server along with any updated information. Thus, execution can restart in the calling server at the point where it was suspended.

This model should have excellent spatial locality properties and good branch predictability as long as the pending work queue is long enough. Moreover, the longer the work queue the better the model is likely to perform because of steadily improving cache locality. Thus, we have an interesting situation where the model may tend to perform poorly under light loads (when there is ample CPU power available) but should steadily improve as additional load is added. These characteristics are in stark contrast to the industry standard model outlined earlier which tends to have the opposite properties.

The diagram on this page outlines how a single server might look in practice.

Let us step back and consider what we are suggesting. The model we are proposing no longer has worker threads like the traditional industry standard model but instead employs a single worker thread per CPU, which is tightly bound to that CPU. Each worker thread no longer works on the behalf of an individual user but rather operates on sets of servers for the benefit of all users. When a worker thread enters a server it is guaranteed that it will never compete for the ownership of shared resources. It is essentially alone with the single goal of processing that servers entire pending work queue. Any other CPU that attempts to operate on the same server in a conflicting way is instead required to find other work on some other server. Thus, the goal is to get every CPU into a different server and allow each of them to work at full speed in isolation on partitioned parts of the overall problem. Recalling our earlier analogy of assembly plant, what we are trying to say is that a worker thread will operate a stage of production line until there is no more work to do. It will then find another part of the production line and work there as well. However, it will never try to work alongside another worker thread on the same stage of the line unless there is guaranteed to be no possibility of conflict.

There is a great deal more to be outlined but before going into additional detail we would like to deal with a few basic objections which we anticipate might be pending at this point. Clearly, this approach turns a lot of basic notions upside down. This is bound to cause some degree of discomfort. A number of obvious objections come to mind such as:

1. Can the complexity be managed?
2. Will the cost of the queue management consume the benefits?
3. Will the pending work queues be long enough?

Let us try to answer these questions in order.

At face value it might appear that using such a model would be a management nightmare. However, careful reflection shows that this is almost a pure object oriented model in that each server is a totally self-contained object with a simple linkage mechanism. A server's job is to provide a well-defined set of services and it is

only allowed access to a restricted part of the overall database state. The concept of having to queue and dequeue pending work packets may seem unpleasant on the surface. However, this is not an issue when it is factored out into a common base class. Moreover, it can be demonstrated that at a fundamental level only a single function needs to be specified for a server. The basic format at the elemental level is something like the following:

VOID ProcessServerCall

```
(  
    CALL_PACKET      *Call,  
    CALL_CONTEXT     *Context  
);
```

Notes: The 'CALL_PACKET' contains all the information about the current outstanding request. The 'CALL_CONTEXT' contains environmental information which may optionally be interesting (e.g., the current CPU).

Another issue is the cost of queuing and dequeuing the work packets on each server. To evaluate this we have constructed two prototypes and have taken various measurements. Our tests indicated that the average cost of calling a server totals less than 400 clocks (running at 1.2 CPI). Obviously this figure is bound to change a little depending on the exact configuration being tested. However, when compared to the total cost of context switching in products like Microsoft SQL Sever 6.5 it is clear that modest savings could reasonably be expected from using our proposed architecture. However, we consider such potential performance improvements insignificant and irrelevant compared to the potential 4-fold speed increase from better cache locality (i.e. reducing 4 CPI to 1 CPI). Hence, we believe this entire issue is mute.

Finally, the question remains: is it reasonable to expect that a server would, on average, get a request queue containing sufficient requests to establish worthwhile cache locality? We believe the answer is affirmative although, once again, a little thought shows that this model has unusual properties. Clearly, when only a single client is connected to a system, such as a database using our methodology, the largest queue size that can be expected would be a single packet. However, with such an extremely light loading we would expect a significant amount of free CPU cycles to be available. Therefore, although the model is

somewhat inefficient it should not matter in this case. It should be noted that this inefficiency relative to the traditional model would probably never be noticed in practice (i.e. an examination of a practical system might indicate 5% CPU usage instead of 3% CPU usage). However, as larger numbers of additional clients are connected (say 6,000) then the characteristics of our proposed architecture start to change. With this loading significant batching would start to occur yielding significantly improved cache locality.

Now let us contrast the above situation to the industry standard model. The industry standard model does very well on a lightly loaded system. However, as the load steadily increases and the threads start to work harder they tend to start contending for global state on the main system bus. In contrast, we note that the new model proposed here would naturally tend to improve as additional load is added whereas the traditional model trends to degenerate. We can deduce from this that there is likely to be a crossing point where both models do equally well, although under a very heavy load it is our contention that our proposed architecture will always do significantly better. We also contend that the new model also naturally encourages a significant degree of parallelism and as we describe in a later section, this further tips the balance in favor of our proposal.

3. Practical Matters

The previous parts of this paper have focused on outlining our alternative architecture for server-oriented applications. From this point onwards we would like to move away from this largely theoretical discussion to a more practical examination of a prototype we developed to test our thesis. We hope to highlight in this discussion some of the opportunities, practicalities and problems of our proposal. The work we shall study is a free standing code base that was developed using C++ and is known internally as the Pipeline Server. Although initially designed as a prototype to support commercial server-oriented applications (i.e., Databases, Internet Servers, and Mail Exchanges), due to its flexibility, the code base has actually been carried forward to see practical service in areas never anticipated by the authors.

We start our examination by revisiting the basic architecture of a server. It was noted in our earlier examination that we passed over some significant areas for the sake of brevity and clarity. We shall now revisit these areas and add additional detail so as to give a more complete picture. One very significant area that was overlooked relates to how servers perform locking. We use this as our starting point.

3.1 Server Locking

In general, selection of locking mechanisms are very significant, as a poor choice can lead to either a lack of concurrency or an excessive number of locking operations. Usually, it is desirable to use a coarse grained locking mechanism as long as this does not adversely effect performance by causing a large number of conflicting operations to block for significantly longer than needed. As we saw earlier our proposed model requires locking to be carried out at a server level so as to permit the maximum degree of cache locality. Unfortunately, locking at this level can cause problems unless targeted alternatives are available. Thus, the Pipeline Server prototype requires each server to be designated as either an Exclusive Server, a Partitioned Server or a Shared Server.

Each of these types implies a different model of locking. A Shared Server guarantees that the code never updates the shared global state. This permits this class of server to be executed on all CPUs at the same time without the need for any locks. Clearly, there may be rare occasions where modifications may be needed. In this case locks must be added to prevent possible corruption. A typical application for this type of server in a database would be for read-only scanning of indexes and data pages (with updates being done by some other class of server). An Exclusive Server is somewhat different, it does permit the shared global state to be modified but requires that the server only execute on a single CPU at any given time. This requirement forces a CPU to acquire a global lock to prevent other CPUs from entering the server. This lock does not cause the other CPUs to be suspended, but rather, forces them to look elsewhere for work. A typical application for this type of server in a database would be for operations like database locks, logging and updates of data or index

pages. Finally, a Partitioned Server also permits the shared global state to be modified but forces all requests to modify specific parts of the global state to a specific CPU (using a supplied predicate). Thus, once again, there can be no resource contention as all requests requiring a particular resource are guaranteed to execute on the same CPU.

3.2. Queues vs. Stacks

In our earlier theoretical discussion we indicated that a server processed requests from a single work queue. This is not the case in Pipeline Server as the basic model is somewhat more sophisticated in current prototype. The most striking difference is that the single work queue discussed in the earlier text is replaced by a collection of work stacks. The number of work stacks matches the number of CPUs available in the system being used. Additionally, each work stack has a matching free stack that holds any available empty work packets. This arrangement was selected for the sake of worthwhile increases in efficiency. Clearly, if a CPU always uses its own private free stack and work stack there is no need to be concerned about contention, inter-processor cache misses or locks. The choice of stacks over queues is appropriate because under a heavy load the front of a queue can easily be lost from the cache. Any subsequent queue traversal can lead to a ripple effect where additional items are also removed from the cache shortly before being referenced, leading to a poor cache-hit ratio. A stack has quite different properties and it is quite reasonable to expect that the top portion of an active stack to be in cache, whatever the system loading. Another notable difference is the addition of multiple global work queues and a single global free queue. The global work queues permit a specific CPU to send work items to some other targeted CPU. These global queues are only used for overall system load balancing and by Partitioned Servers. Each CPU is required to check its global work queue every time it executes a server. Any work found is completed and returned to the calling CPU as appropriate. The global free queue has a different purpose and contains an additional supply of free work packets for the associated server. This allows a CPU to extract additional free work packets as needed and return them later if it detects excessive inventories. The global queues are

generally operated on with batch operators (i.e., add or remove 16 work packets per call) to reduce contention and cross CPU communication. It is worth noting that only the queues have locks associated with them and that they are the sole mechanism for moving work packets between CPUs. The attributes of queues are advantageous in this situation, as they tend to reduce the number of inter-CPU cache faults by permitting the work packets to age before they are extracted by the target CPU.

Let us now expand on our earlier diagram and examine the architecture of the Pipeline Server prototype in more detail.

This expanded diagram is significantly different from our earlier version. Note the additional global free queue, the additional two global work queues (1 per CPU) and the new local free stacks and work stacks.

The Pipeline Server prototype requires each CPU to explicitly manage all the work packets it uses. This ensures that work packets can never implicitly move between CPUs. Consequently, from a high level perspective the Pipeline Server fully partitions its workload even when running on an SMP system. This partitioning significantly increases the scalability, as there is little CPU cross talk. Measurements have shown that the core typically consumes around 3% of an SMP systems bus bandwidth even under a heavy load (i.e. 1,000 users on a 4x200Mhz Pentium Pro with a 256k L2 cache executing at 800M instructions per second running a simple test application).

The Pipeline Server prototype attempts to take care of a large number of issues so as to make the implementation of servers as simple as possible. A collection of typical issues, such as claiming the appropriate server locks and dynamic load balancing, are automatically dealt with by the core engine. The image presented to a server developer is essentially a simple single threaded state machine (with some additional restrictions and rules). This allows the server writer to focus on the domain specific problem rather than the usual multitude of unrelated implementation issues such as potential race conditions. This simplification of the development environment often leads to worthwhile reductions in code complexity that in turn can even further enhance performance.

4. Parallel Execution

A significant topic not covered thus far is that of Parallel Execution. We would like to show that using the Pipeline Server model it is possible to produce a form parallelism that has dramatically different characteristics than traditional parallel programming architectures. In particular, we will show that significant amounts of parallelism (perhaps too much) can be produced and managed at little or no additional cost.

The traditional parallel programming architecture is usually based on either processes or threads. In this model multiple threads (or processes) are created and executed either serially or in parallel on one or more CPUs. Unfortunately, threads (and processes) are a fairly heavy weight entities and can not be rapidly created, context switched, or destroyed. This is not solely due to Operating System (OS) overhead but also due to the fact that threads (and processes) often save important data on their private stack. The Pipeline Server model approaches this problem in a totally different way. As we saw earlier, the model encourages an individual server to have a queue of pending requests. Typically these work packets are from different clients. However, there is no reason a server can not process multiple requests for the same client or even multiple requests from multiple clients mixed together. A server always considers each request in isolation regardless of what any client (including the current client) might be doing elsewhere.

Now let us consider the implications of this adjustment to our basic model. As an example we will consider how a typical database might scan an index looking for matching keys. For simplicity, we shall assume there is no background load and that only a single packet is waiting to be processed. This initial work packet points at the base of an index and requests that the root node be searched for any key that matches the predicate LIKE "N%". When the appropriate server is executed it will scan the root index page for matches. Assume, for the sake of the example, it finds 2 matches. Each of these matches refers to another index page so we can create a new work packet for each match and queue it on the current server for processing later. When the server has completed scanning the root index page it will take the first of these newly added packets and process it. Let's

assume that the server finds that this index page is not in RAM. It will queue a request on another server to read the page from disk. It will then move on to process the second work packet. This index page might identify a data page to examine. This request is then passed to another server to process the data page.

Let us review what the above explanation has exposed. We saw that the scan of the initial root page generated 2 hits, which were queued back on the same server. Another way of saying this is that we identified and exposed 2 parallel execution paths. We noted that no special code was required to execute these parallel paths and that the creation of the additional parallelism was cheap (i.e. the cost of creating an additional work packet). Next, we saw that one of the parallel paths became blocked because the index page was not available. However, the second parallel path was able to continue and located one of the target data pages immediately. Another way of stating this is that we saw out-of-order execution. That is, we were able to continue on all unrelated execution paths.

Clearly, it is not possible to explain all the situations we can apply this type of methodology. However, it is hoped that the reader will perceive that this notion radically changes the traditional parallel processing problems. The common issues of exposing and managing parallelism are somewhat simplified and replaced by new issues of managing the potentially massive amount of parallelism and out-of-order execution available within the system. We believe this is a desirable situation, as any additional work tends to enhance the performance of our model by extending the average work queue length which in turn improves the overall the cache locality of the servers. Moreover, in many ways the traditional metrics are stood upside-down and an array of new opportunities become available.

The Pipeline Server model totally changes the way a typical database query would be executed. Many of the traditional metrics are radically changed. The cost of managing large degrees of parallel execution is significantly reduced. Some of the traditional services in applications like databases simply become obsolete. As an example, it can be seen that there is no longer any need for a read-ahead manager as an exact list of required pages can quickly be generated and the I/O initiated (even if the query is using a

secondary index to locate the data). Moreover, any isolated data pages already in memory will typically be identified, processed and returned immediately instead of lying fallow and possibly being pushed out of memory by incoming data (only to be re-read again later).

The Pipeline Server model of parallel execution was tested within SQL Server 6.5 in the area of index verification. The results obtained were somewhat encouraging. The modifications made were limited in nature and scope but produced an increase in performance of 900%. Obviously the vast majority of this speed up was due to the additional parallelism. However, the CPI measurements dropped from just over 3 CPI to around 2 CPI. This was a reversal of the usual trend within SQL Server. Related stress testing showed that the algorithm was very tolerant, showing little degradation even when forced to perform a task with a fan out of 1,000 parallel contexts. These tests lead us to believe that the essential methodology is sound and generally applicable.

5. Dynamic Load Balancing

We outlined the methodology used to support parallelism in the Pipeline Server prototype in the previous section. We would now like to move on and examine the related issue of dynamic load balancing. Let us begin with a hypothetical example illustrating the sort of problems encountered in a commercial computing environment. Our illustration will center on a large parallel query submitted to a relational database at 7:30am (expected run time 2 hours). At this time in the morning the target system happens to be largely idle and therefore selecting the maximum degree of parallelism seems a good choice. Unfortunately, by around 8:30am most of the early shift has arrived and the system loading is approaching its daily peak. Thus, it appears that the minimum degree of parallelism might have been a better choice. However, at 9:00am a staff meeting is called and the loading rapidly tails off for 30mins after which time it rapidly climbs back to peak loading again.

The above illustration highlights that it is hard to predict dynamic loading in a typical commercial computer system due to a range of external factors (i.e. meetings, sickness and weather). Therefore, if we are unable to statically plan, the

only reasonable alternative is to balance the dynamic loading. Although this area is traditionally in the domain of the Operating Systems (OSs) we would like to introduce an alternative methodology for doing dynamic load balancing for our Pipeline Server prototype.

We are unlikely to be able to dynamically balance the loading and maximize system throughput unless we have some basic controls. Clearly, we will need to be able to move work between resources (i.e. from one CPU to another) and control the amount of parallelism (i.e. to maximize the throughput but prevent a small number of users from hogging the system). We shall examine mechanisms for these operations in the following two paragraphs.

Due to the dynamic nature of user requests CPUs can become unevenly loaded. The Pipeline Server prototype overcomes this problem by initiating dynamic load balancing every 25 milliseconds. The load balancing thread collects information about the outstanding work packets on every CPU and server. It calculates how many of these work packets would need to be reassigned to alternative CPUs to bring the loading into perfect balance. It then instructs each overloaded CPU to send a specific number of work packets to underloaded CPUs. This merely requires the work packets to be moved to the global work queue of the target CPU (with a special flag). The transferred work is not special in any way and is treated as though it had been assigned to the target processors. Normal processing is not affected and a balanced loading is immediately restored (even if the previous imbalance was quite severe). Although this algorithm is somewhat crude various testing has shown it to be highly effective. The amount of work transferred is typically minimal (given that new work is normally allocated to CPUs in a round robin fashion).

The dynamic control of parallel execution is traditionally a complex task. However, the Pipeline Server prototype is able to naturally deal with such problems due to the design of its servers. As we saw earlier selected operations like searching an index can produce significant additional parallelism. This additional parallelism is controlled by a simple quota algorithm applied each time additional parallelism is exposed. The quota algorithm sets the following limit on the amount of parallelism allowed for each user: (((Number of System

CPU's * 1,000) / (Number of Clients) + 1). If a client has enough quota, each opportunity for parallelism will be taken. If not, each opportunity will be declined but noted for later. Thus, if a single client is logged on to a dual CPU system, any task is permitted to execute up to 2,000 way parallel. As additional clients log on, the degree of parallelism permitted is reduced. When 2,000 or more clients are logged on then each client is limited to a single active work packet. As additional parallelism typically only has a limited life span, any dramatic increase in the loading suppresses all additional parallelism until the loading is brought under control (i.e. typically less than 2 seconds). If the loading falls dramatically, every opportunity to increase the parallelism is taken steadily increasing the system loading.

The above algorithms are still being enhanced at the time of writing, but it is hoped that the reader can see that our proposed model radically simplifies the traditional problems of load balancing. Moreover, modest overloading does not cause inefficiency it merely encourages the CPU's to work harder (due to the longer work queues and consequent improved cache locality). Even with severe overloading, all outstanding work is guaranteed to make some forward progress. This is ensured because each server is required to empty its entire pending work queue, regardless of which user submitted the work packet. Thus, some users may observe more progress than others (due to extra parallelism) but this is only temporary, as the loading will swiftly move towards uniformity due to the load balancing algorithms.

6. Conclusion

The evolutionary changes in hardware have slowly but surely eroded the validity of a number of traditional software architectures and metrics. Hallowed metrics such as counting instructions are no longer a good measure of efficiency or performance. A new set of metrics such as cache misses, clocks, mis-predicted jumps and stalls better reflect the current hardware realities.

In this paper we have proposed radically new software architecture focused on enhancing performance by improving spatial locality. The proposed architecture rejects much traditional thinking and sets out to overcome a number of

notable problems in areas such as dynamic load balancing, parallelism, scalability. We hope that the reader agrees with us that our suggestion is a significantly different approach to building server-oriented applications. We further hope that it will be found useful in the unending battle to build ever bigger, better and faster computer systems.

Acknowledgements

I would particularly like to thank Charles Levine, Jim Gray and Mike Jones for their help with this paper and for their patience. Due to work pressures Rick and I were not always able to polish each revision as well as we should have. Sincere thanks.